Caches 3

Prof. Hakim Weatherspoon

CS 3410, Spring 2015

Computer Science

Cornell University

See P&H Chapter: 5.1-5.4, 5.8, 5.10, 5.15; Also, 5.13 & 5.17

Overview

Writing to caches: policies, performance

Cache tradeoffs and performance

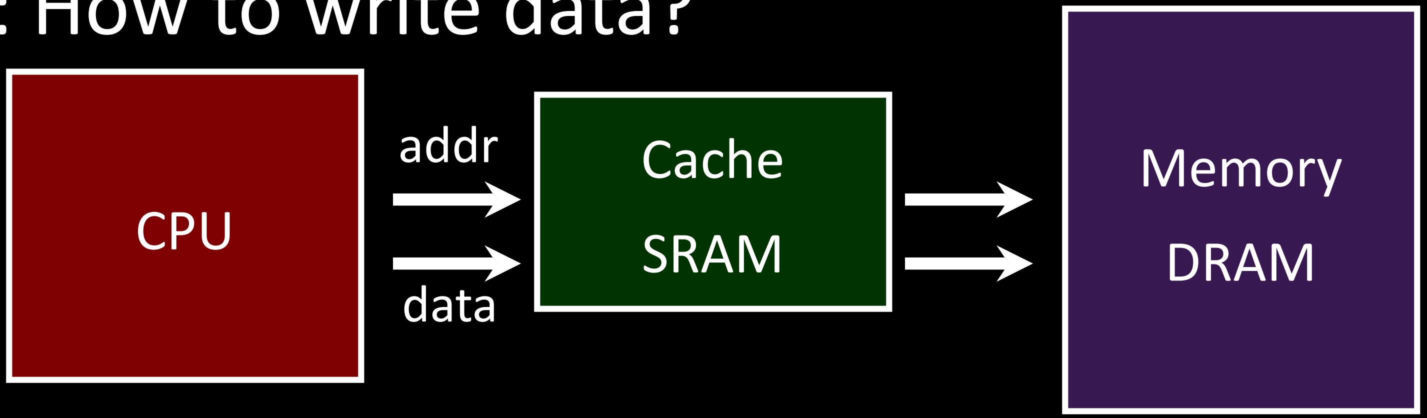
What about Stores?

Where should you write the result of a store?

- If that memory location is in the cache?
  - Send it to the cache
  - Should we also send it to memory right away?
(write-through policy)
  - Wait until we evict the block (write-back policy)
- If it is not in the cache?
  - Allocate the line (put it in the cache)?
(write allocate policy)
  - Write it directly to memory without allocation?
(no write allocate policy)

Cache Write Policies

Q: How to write data?



If data is already in the cache…

No-Write

writes invalidate the cache and go directly to memory

Write-Through

writes go to main memory and cache

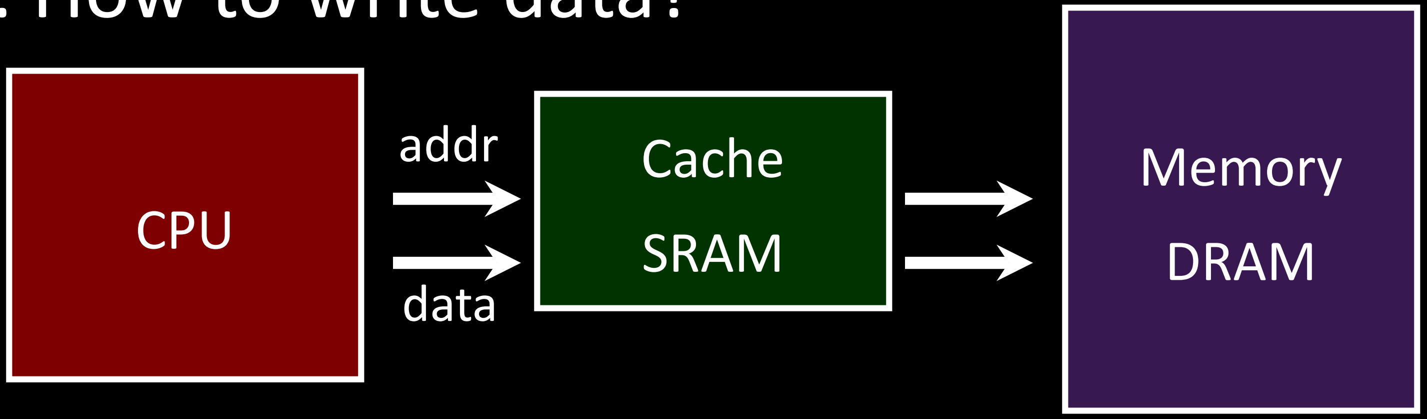
Write-Back

CPU writes only to cache

cache writes to main memory later (when block is evicted)

Write Allocation Policies

Q: How to write data?



If data is not in the cache…

Write-Allocate

allocate a cache line for new data (and maybe write-through)

No-Write-Allocate

ignore cache, just go to main memory

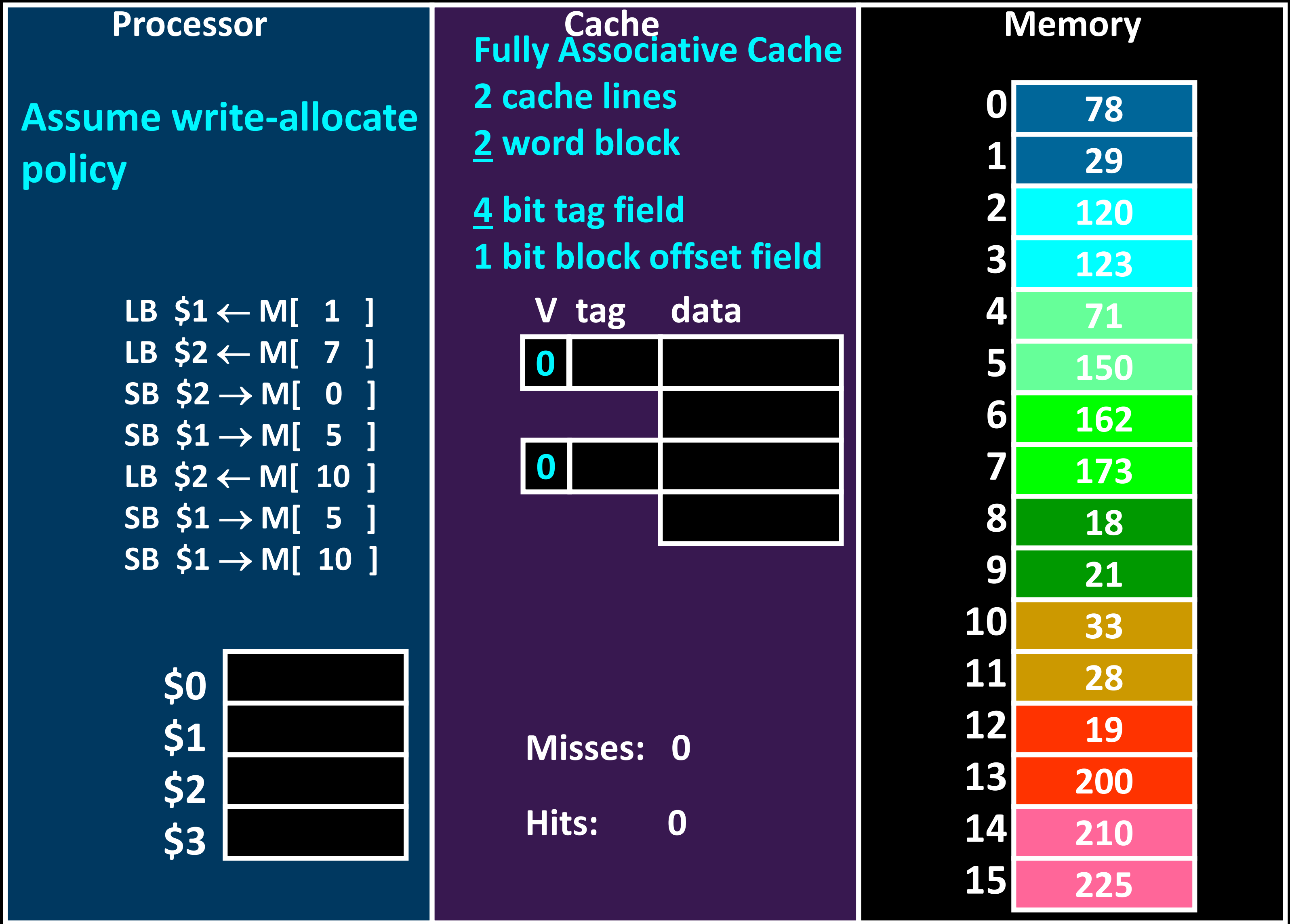
Next Goal

How does a write-through cache work?

Assume write-allocate

Handling Stores (Write-Through)

Using **byte addresses** in this example! Addr Bus = 5 bits



Processor

Assume write-allocate policy

```
LB  $1 ← M[  1  ]
LB  $2 ← M[  7  ]
SB  $2 → M[  0  ]
SB  $1 → M[  5  ]
LB  $2 ← M[ 10  ]
SB  $1 → M[  5  ]
SB  $1 → M[ 10  ]
```

$0	
$1	
$2	
$3	

Cache

Fully Associative Cache

2 cache lines

2 word block

4 bit tag field

1 bit block offset field

V	tag	data
0		
0		

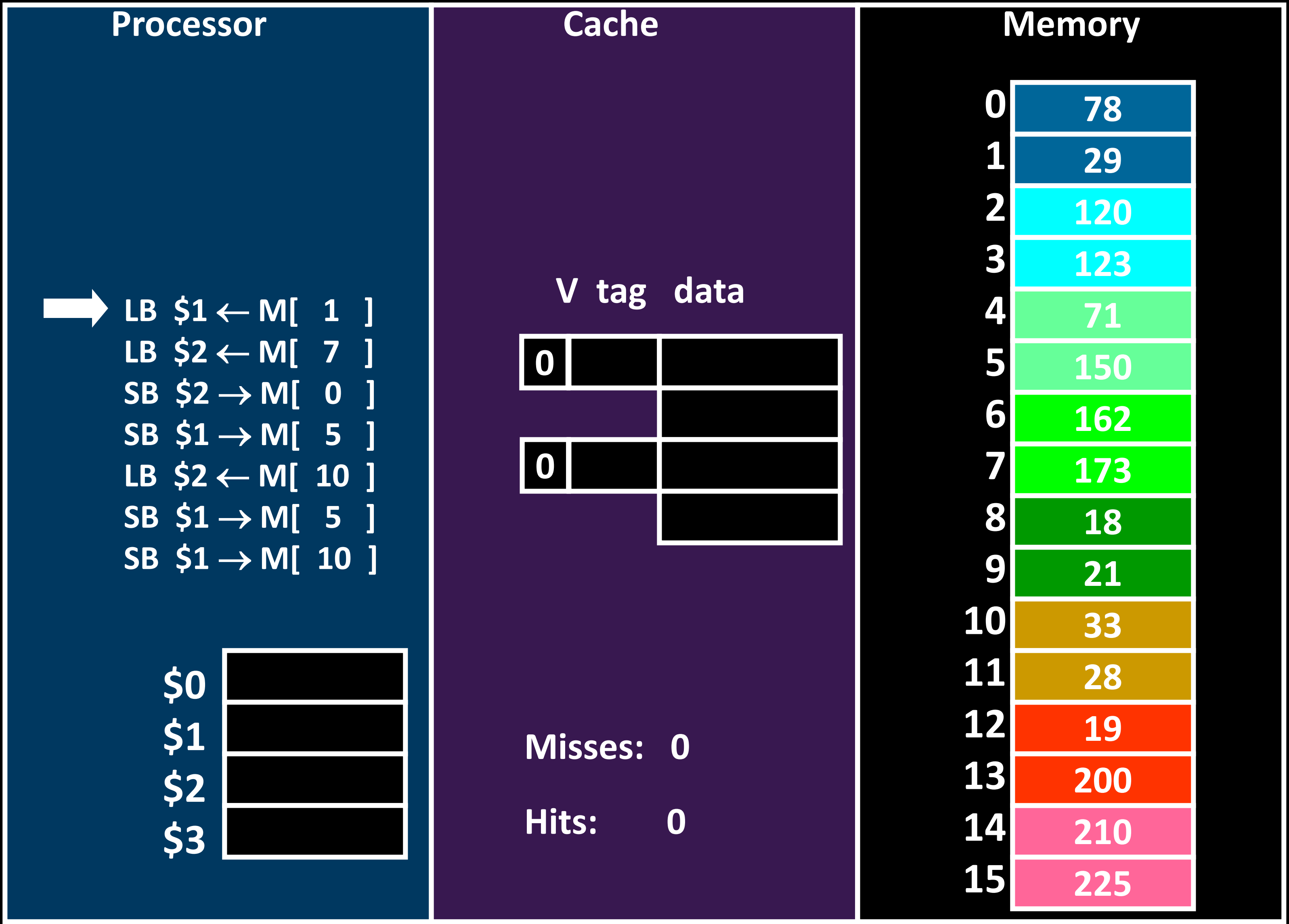
Misses: 0

Hits: 0

Memory

0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Write-Through (REF 1)



Processor

```
➡ LB  $1 ← M[  1  ]
   LB  $2 ← M[  7  ]
   SB  $2 → M[  0  ]
   SB  $1 → M[  5  ]
   LB  $2 ← M[ 10  ]
   SB  $1 → M[  5  ]
   SB  $1 → M[ 10  ]
```

$0
$1
$2
$3

Cache

V	tag	data
0		
0		

Misses: 0

Hits: 0

Memory

0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

How Many Memory References?

Write-through performance

Each miss (read or write) reads a block from mem

Each store writes an item to mem

Evictions don’t need to write to mem

Summary: Write Through

Write-through policy with write allocate

- Cache miss: read entire block from memory
- Write: write only updated item to memory
- Eviction: no need to write to memory

Write-Through vs. Write-Back

Can we also design the cache NOT to write all stores immediately to memory?

- Keep the most current copy in cache, and update memory when that data is evicted (write-back policy)
- Do we need to write-back all evicted lines?
  - No, only blocks that have been stored into (written)

Write-Back Meta-Data

V	D	Tag	Byte 1	Byte 2	… Byte N

V = 1 means the line has valid data

D = 1 means the bytes are newer than main memory

When allocating line:

- Set V = 1, D = 0, fill in Tag and Data

When writing line:

- Set D = 1

When evicting line:

- If D = 0: just set V = 0
- If D = 1: write-back Data, then set D = 0, V = 0

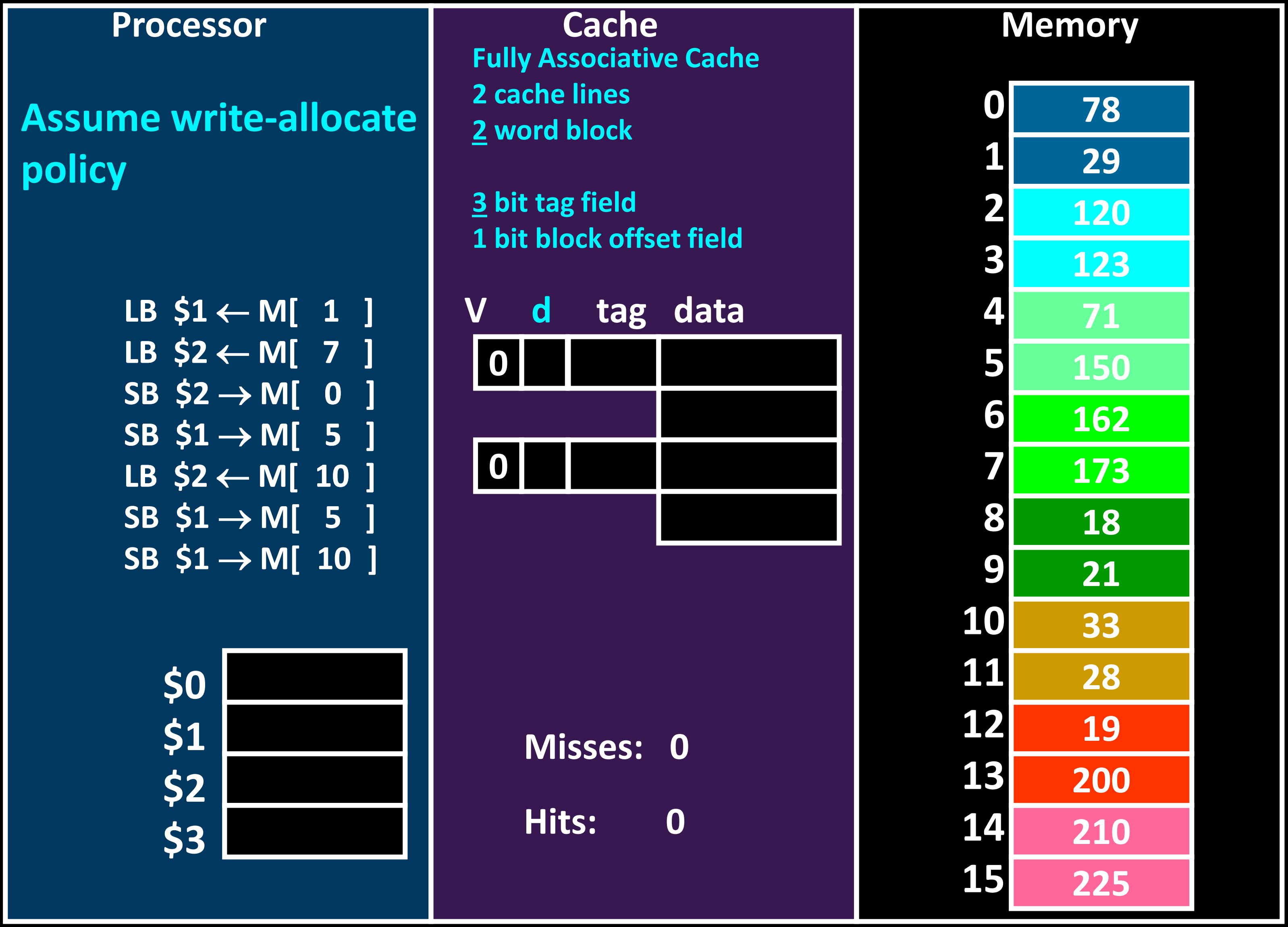
Write-back Example

Example: How does a write-back cache work?

Assume write-allocate

Handling Stores (Write-Back)

Using **byte addresses** in this example! Addr Bus = 5 bits



Processor

Assume write-allocate policy

LB $1 ← M[1]
LB $2 ← M[7]
SB $2 → M[0]
SB $1 → M[5]
LB $2 ← M[10]
SB $1 → M[5]
SB $1 → M[10]

Register	
$0	
$1	
$2	
$3	

Cache

Fully Associative Cache
2 cache lines
2 word block

3 bit tag field
1 bit block offset field

V	d	tag	data
0			
0			

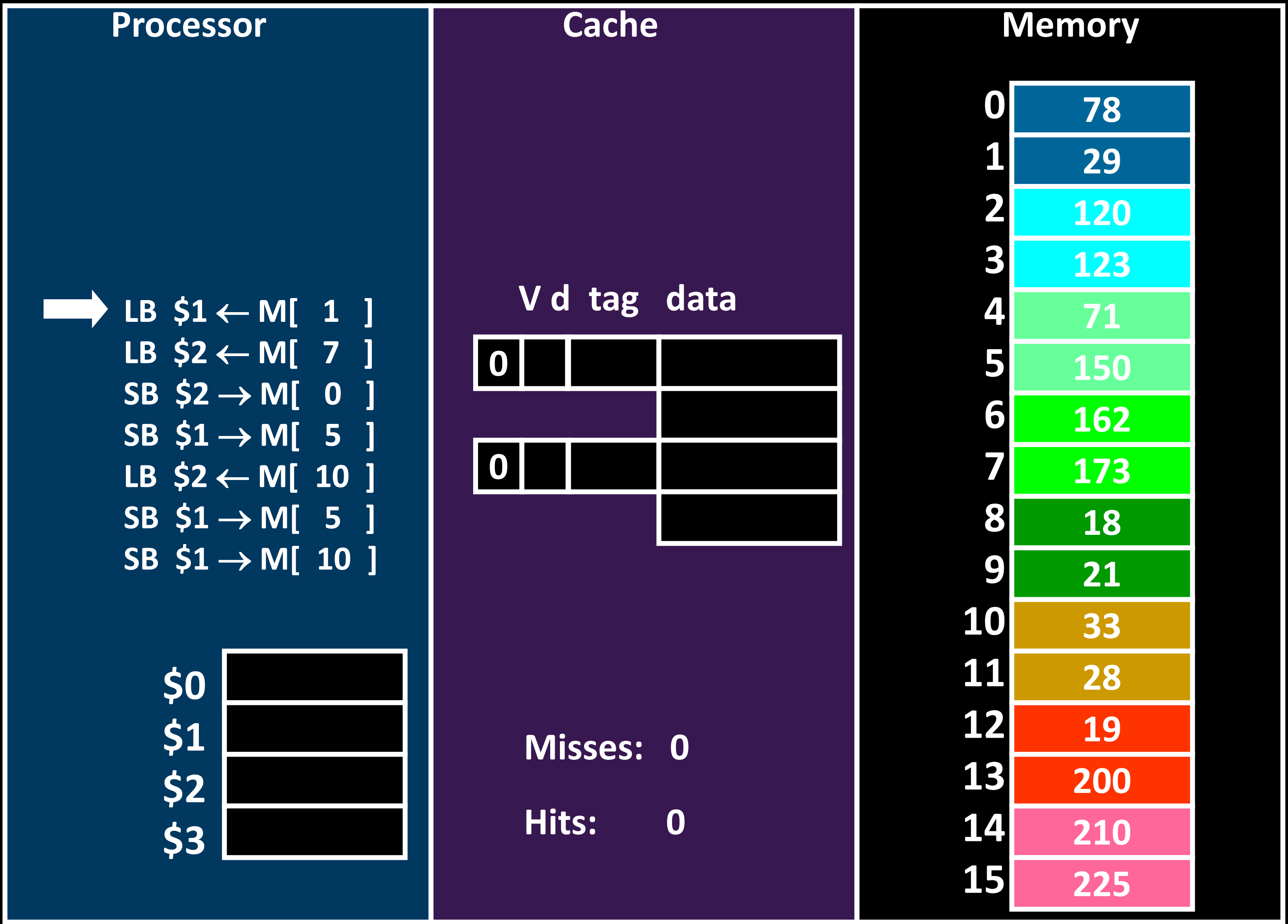
Misses: 0

Hits: 0

Memory

0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Write-Back (REF 1)



Processor

```
➡ LB  $1 ← M[  1  ]
  LB  $2 ← M[  7  ]
  SB  $2 → M[  0  ]
  SB  $1 → M[  5  ]
  LB  $2 ← M[ 10  ]
  SB  $1 → M[  5  ]
  SB  $1 → M[ 10  ]
```

$0	
$1	
$2	
$3	

Cache

V	d	tag	data
0			
0			

Misses: 0

Hits: 0

Memory

0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

How Many Memory References?

Write-back performance

Each miss (read or write) reads a block from mem

Some evictions write a block to mem

So is write back just better?

What are other performance tradeoffs between write-through and write-back?

How can we further reduce penalty for cost of writes to memory?

Performance: An Example

Performance: Write-back versus Write-through

Assume: large associative cache, 16-byte lines

```
for (i=1; i<n; i++)
           A[0] += A[i];
```

```
for (i=0; i<n; i++)
           B[i] = A[i]
```

Performance Tradeoffs

Q: Hit time: write-through vs. write-back?

Q: Miss penalty: write-through vs. write-back?

Write Buffering

Q: Writes to main memory are **slow!**

Q: When does it help?

Write-through vs. Write-back

Write-through is slower

- But simpler (memory always consistent)

Write-back is almost always faster

- write-back buffer hides large eviction cost
- But what about multiple cores with separate caches but sharing memory?

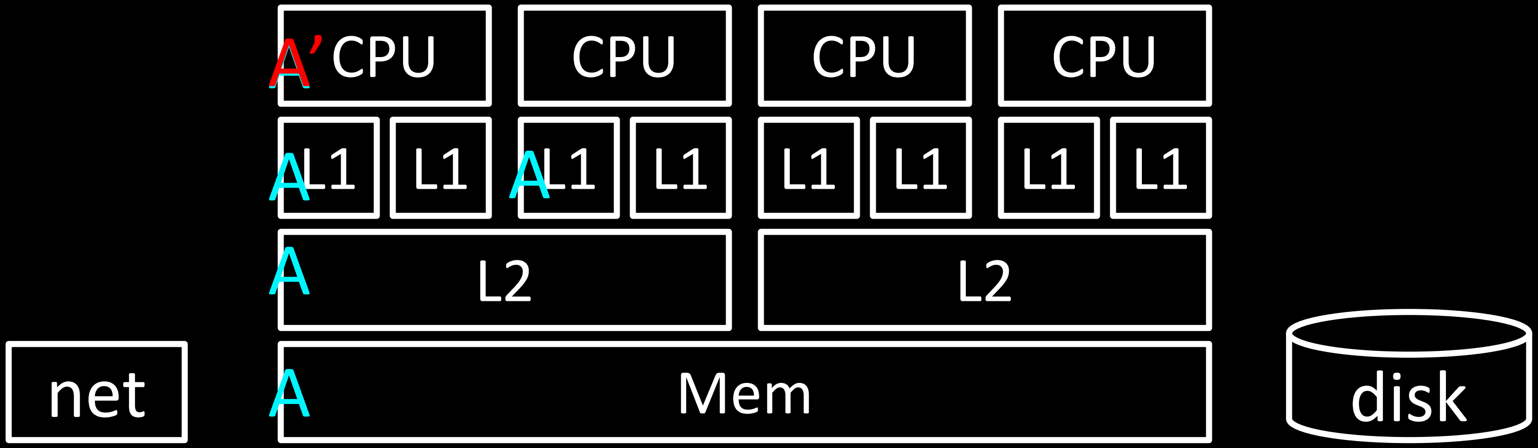
Write-back requires a cache coherency protocol

- Inconsistent views of memory
- Need to “snoop” in each other’s caches
- Extremely complex protocols, very hard to get right

Cache-coherency

Q: Multiple readers and writers?

A: Potentially inconsistent views of memory



Cache coherency protocol

- May need to snoop on other CPU’s cache activity
- Invalidate cache line when other CPU writes
- Flush write-back caches before other CPU reads
- Or the reverse: Before writing/reading…
- Extremely complex protocols, very hard to get right

Summary: Write Through

Write-through policy with write allocate

- Cache miss: read entire block from memory
- Write: write only updated item to memory
- Eviction: no need to write to memory
- Slower, but cleaner

Write-back policy with write allocate

- Cache miss: read entire block from memory
  - But may need to write dirty cacheline first
- Write: nothing to memory
- Eviction: have to write to memory, entire cacheline because don't know what is dirty (only 1 dirty bit)
- Faster, but complicated with multicore

Next Goal

Performance: What is the average memory access time (AMAT) for a cache?

AMAT = %hit x hit time + % miss x miss time

Cache Performance Example

Average Memory Access Time (AMAT)

Cache Performance (very simplified):

L1 (SRAM): 512 x 64 byte cache lines, direct mapped

Data cost: 3 cycle per word access

Lookup cost: 2 cycle

16 words (i.e. 64 / 4 = 16)

Mem (DRAM): 4GB

Data cost: 50 cycle for first word, plus 3 cycles per subsequent word

Cache Performance Example

Average Memory Access Time (AMAT)

Cache Performance (very simplified):

L1 (SRAM): 512 x 64 byte cache lines, direct mapped

Data cost: 3 cycle per word access

Lookup cost: 2 cycle

16 words (i.e. 64 / 4 = 16)

Mem (DRAM): 4GB

Data cost: 50 cycle for first word, plus 3 cycles per subsequent word

Multi Level Caching

Cache Performance (very simplified):

L1 (SRAM): 512 x 64 byte cache lines, direct mapped

Hit time: 5 cycles

L2 cache: bigger

Hit time = 20 cycles

Mem (DRAM): 4GB

Hit rate: 90% in L1, 90% in L2

Often: L1 fast and direct mapped, L2 bigger and higher associativity

Performance Summary

Average memory access time (AMAT)

depends on cache architecture and size

access time for hit,

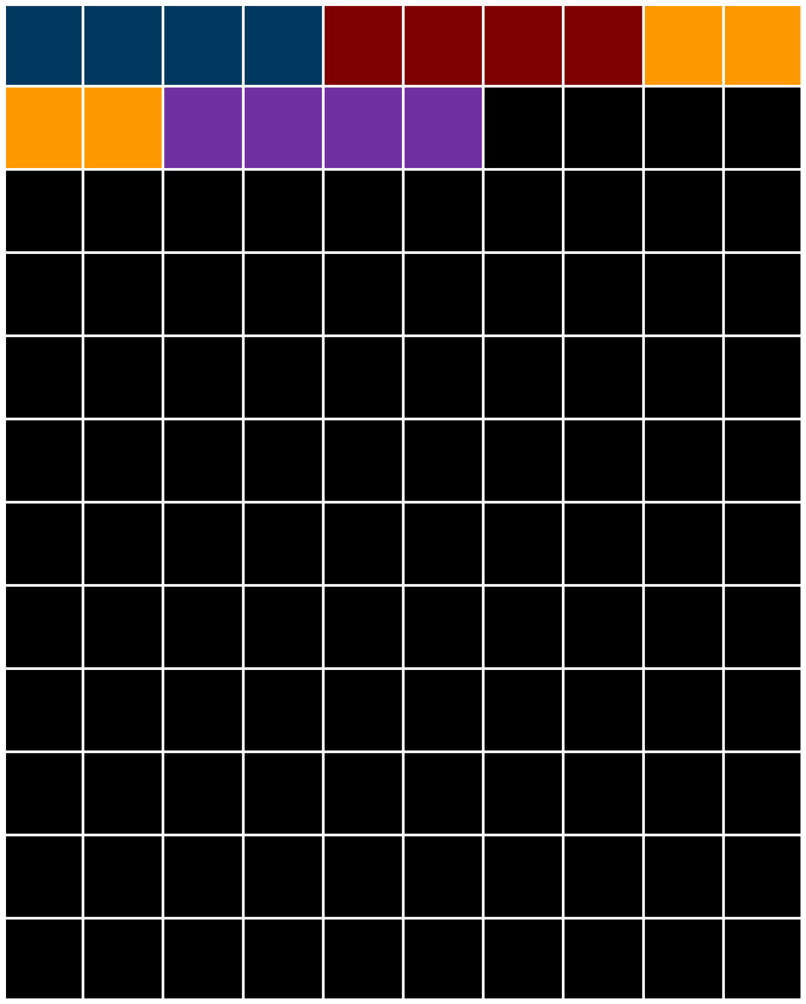
miss penalty, miss rate

Cache design a very complex problem:

- Cache size, block size (aka line size)
- Number of ways of set-associativity (1, N, ∞)
- Eviction policy
- Number of levels of caching, parameters for each
- Separate I-cache from D-cache, or Unified cache
- Prefetching policies / instructions
- Write policy

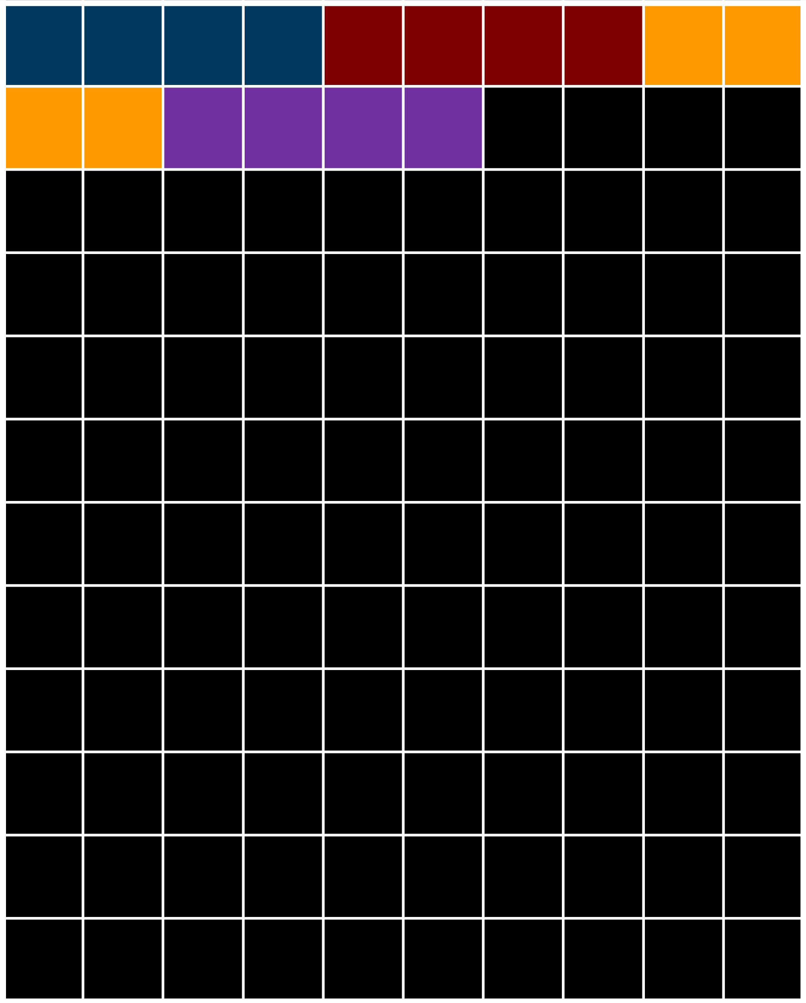
Cache Conscious Programming

```
// H = 12, W = 10
int A[H][W];

for(x=0; x < W; x++)
   for(y=0; y < H; y++)
      sum += A[y][x];
```



Cache Conscious Programming

```
// H = 12, W = 10
int A[H][W];

for(y=0; y < H; y++)
   for(x=0; x < W; x++)
      sum += A[y][x];
```



A Real Example

Dual-core 3.16GHz Intel
(purchased in 2011)

```
> dmidecode -t cache
Cache Information
        Configuration: Enabled, Not Socketed, Level 1
        Operational Mode: Write Back
        Installed Size: 128 KB
        Error Correction Type: None
Cache Information
        Configuration: Enabled, Not Socketed, Level 2
        Operational Mode: Varies With Memory Address
        Installed Size: 6144 KB
        Error Correction Type: Single-bit ECC
> cd /sys/devices/system/cpu/cpu0; grep cache/*/*
cache/index0/level:1
cache/index0/type:Data
cache/index0/ways_of_associativity:8
cache/index0/number_of_sets:64
cache/index0/coherency_line_size:64
cache/index0/size:32K
cache/index1/level:1
cache/index1/type:Instruction
cache/index1/ways_of_associativity:8
cache/index1/number_of_sets:64
cache/index1/coherency_line_size:64
cache/index1/size:32K
cache/index2/level:2
cache/index2/type:Unified
cache/index2/shared_cpu_list:0-1
cache/index2/ways_of_associativity:24
cache/index2/number_of_sets:4096
cache/index2/coherency_line_size:64
cache/index2/size:6144K
```

A Real Example

Dual-core 3.16GHz Intel (purchased in 2009)

Dual 32K L1 Instruction caches

- 8-way set associative
- 64 sets
- 64 byte line size

Dual 32K L1 Data caches

- Same as above

Single 6M L2 Unified cache

- 24-way set associative (!!!)
- 4096 sets
- 64 byte line size

4GB Main memory

1TB Disk

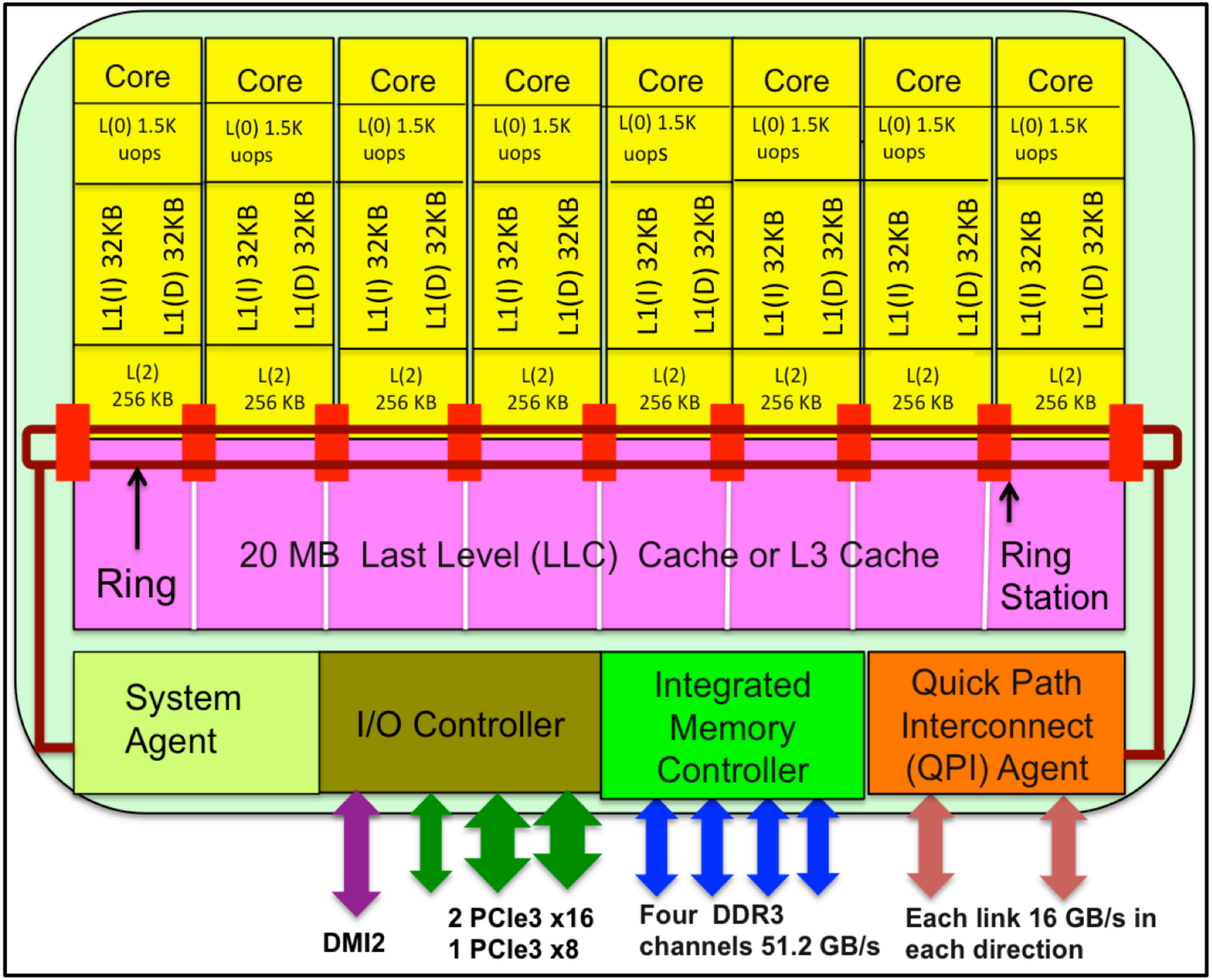


Figure 1. Schematic diagram of a Sandy Bridge processor.

Summary

Memory performance matters!

- often more than CPU performance
- … because it is the bottleneck, and not improving much
- … because most programs move a LOT of data

Design space is huge

- Gambling against program behavior
- Cuts across all layers:
  users → programs → os → hardware

Multi-core / Multi-Processor is complicated

- Inconsistent views of memory
- Extremely complex protocols, very hard to get right